Documentation of the Graph Class Library (GCL)

by

Richard S. Stevens

November 3, 2001

1. Introduction

At NRL (Naval Research Laboratory) we wrote the PGM (Processing Graph Method) Specification [1] to describe the basic features of a support system for developing applications for a distributed network of processors. In the PGMT (PGM Tool) project we developed an implementation of PGM. The GCL (Graph Class Library) is one component of PGMT.

In this document we attempt to capture the overall design and structure of the GCL, so that others may understand the code and improve on it. We wrote this document after the GCL code was written and tested. There was no formal design document or collection of design documents for the entire GCL. Instead, there were individual design documents for some parts of the GCL. Other parts were designed "on the fly" after discussions with others on the PGMT team.

In addition to capturing the overall design, we also strive to capture the intuition, insight, and motivation behind that design. We also include some discussion of the other components of PGMT, so that the reader may better understand the interfaces between the GCL and those other components.

We assume that the reader is familiar with the PGM 2.1 Specification [1]. We also assume that the reader is familiar with the ANSI standard C++ programming language, including the Standard Template Library (STL).

1.1. PGMT Structure

This sub-section gives an overview of the modular structure of PGMT.

A user application consists of a *Command Program* and a *Graph*, each of which is the instance of a respective C++ class.

PGMT provides most of the class definition for the Command Program. Although this class definition is intended to be generic for all applications, the user may modify it. The user must provide the application specific part of the command program, which includes, but is not necessarily limited to, the body of C++ code that makes up the run() method of the Command Program class. This code must include a call to the graph constructor. The specifics are provided in other PGMT documentation. This document being concerned primarily with the implementation of PGM Graphs, we do discuss Command Programs in this document only in the context of the interface between the Command Program and Graph.

We describe briefly the following modules of PGMT:

1.1.1. Graph User Interface (GUI)

PGMT provides a GUI for developing PGM graphs. Using standard "point-and-click" techniques, the user may sit at a computer workstation and draw his PGM graph. Menus are provided; the user places icons on the screen to represent nodes on the graph and connects them with arrows. The user fills out forms with parameters for the inputs and outputs of the graph and

for the nodes of the graph. The GUI saves a file specification of a PGM graph in an ASCII file called the GSF (Graph State File) [2]. Coupled with the GUI is a translator that generates C++ code for the user's graph class definition from the GSF [3].

1.1.2. Middleware

PGMT is designed to run on a distributed network of processors. We use the standard MPI (Message Passing Interface) for communication between processors. To encapsulate the specifics of MPI and other architecture specific aspects of the network, we have the *Middleware* Module, which supports communication for the GCL [6] as well as some other PGMT modules. The idea is that if we should decide to use a different communication tool set, it will be necessary only to rewrite the Middleware Module.

In the remainder of this document, we use the words *process* and *processor* consistently with the use of these terms in MPI. Specifically, a *processor* is physical machine, and a *process* is a virtual processor. One may assign several processes to the same processor. All the processes assigned to the same processor act like separate threads. They are time shared on that processor and communicate with each other just as if they were assigned to separate processes. For the programmer using MPI, this distinction is transparent, as MPI is concerned only with processes and not with processors. Henceforth will speak of a *processor* only in the context of a physical machine. We will usually speak of *processes* in the context of inter-process communication.

In addition to providing the interface to MPI, the Middleware defines MTOOL. The purpose is to support user-defined data types. To pass data between processes, an MPI type must be defined for each user-defined data type. In the past, the user must define the MPI type manually. In PGMT, MTOOL automates the definition of the MPI type from the user-written definition of the data type.

1.1.3. PEP (PGMT Execution Program)

The PEP is responsible to decide how transitions should be initially assigned to processes and how they should be reassigned during graph execution. Within each process, the PEP keeps track of transitions that are ready to execute and schedules them for execution. The interface between the GCL and PEP is specified in [5].

1.1.4. GCL (Graph Class Library)

The GCL is the core of PGMT. The GCL defines all the classes to support the construction and execution of the user-defined graph. This includes families of nodes, families of included graphs, data tokens, the base classes for user-defined ordinary transitions, determining the readiness of transitions to execute, the sending and receiving of messages between processes, and the implementation of assignment and reassignment of transitions to processes. The GCL has interfaces with all of the other modules described above [3, 6, 5].

1.2. Organization Of This Document

In the next major section we discuss the implementation of families in the GCL. We devote a major section to this topic because of its ubiquitous presence in PGM, which reflects its

importance in the PGM Spec [1]. Much of the rest of this document depends on an understanding of how families are implemented in GCL.

Following that, we discuss the structures used in the construction of a graph. Specifically, we will define structures used to construct and maintain families of nodes, families of ports, and families of included graphs. In so doing, we will discuss the construction of the nodes in the graph and how they are connected. After describing graph construction, we include a section on graph execution.

We devote a section to distributed processing with nodes that are semi-permanently assigned to processes. Reassignment of nodes to processes is accomplished only when necessary to improve performance. We strove to accomplish reassignment of nodes to processes with little unnecessary interruption of graph execution.

We then discuss our approach to testing and validation of the GCL. We also describe the test environment and how to use it.

Finally, we have a section on future work that includes some additional features that we feel would enhance the utility and performance of PGMT.

Following the list of references, we provide a number of appendices. Appendix A gives a list of the classes, including the class hierarchy, and a description of the methods and how and when they are called. Appendix B gives the design of the Pack Transition. Appendix C gives an action matrix for reassignment of places to processes. Appendix D discusses "switch" and "merge", which we feel would enhance the usefulness of PGMT.

1.3. Apologies

Before we proceed, we say a few words about the context in which we developed the GCL, which may provide some insight into its current structure.

We decided early in the project to use the C++ programming language. The primary factors leading to this decision were the following:

- We needed an object oriented language to facilitate design and implementation.
- The requirement to support user-defined data types implied the need for a language that supports user-defined data types.
- We needed a language that would be easily compiled to produce executable code that runs fast.
- We needed a standardized language.
- We needed a language with a compiler that was available for free.

These requirements led us to choose C++ with templates, including the Standard Template Library (STL), as the language, using the GNU g++ compiler from the Free Software Foundation (FSF). We recognized at the time we made this decision that C++ is not the easiest programming language to write.

Throughout the project, our budget was limited, and we were pressured to produce a version of PGMT that could be demonstrated as early as possible. In addition, while we had some

experience programming in C++, our knowledge of the language was less than complete. We tended to design and code using the features with which we were familiar. When we wanted to do something that needed a feature of the language that was new to us, we read the books. Often we learned things that would have helped earlier. Time and schedule constraints prevented us from going back and rewriting what we had already done. Thus the style of coding is not consistent throughout, reflecting our learning curve during the course of the project.

Even today, now that the GCL is complete to the extent we planned at the beginning, our knowledge of C++ is not complete. Thus even the most recently written code is in a style that is less than ideal.

Another kind of inconsistency arises from our early attempt to use a graphic design tool, based on the UML (Unified Modeling Language) [7], to generate both documentation and boilerplate C++ code automatically. After using this tool for awhile we found that it was impeding our progress, and we therefore dropped it. After we dropped the tool, we adopted a coding style that is not consistent with the tool. However we did not go back and change the code that the tool generated in our earlier work.

Our definition and use of exceptions is very crude. Each exception announces its problem without giving any indication of where or how the problem occurred. In addition, no stack trace is given to indicate the method call hierarchy. There are too many different exceptions, each of which we defined as we went along. With additional time we would rewrite all the exceptions with care and attention to providing useful diagnostic information. Specifically, we would do this so that when an exception is thrown, the user-assigned name of the object throwing the exception is announced (if it has a name), and a method stack trace is given. We will say more about this in a later section, where we list items that we recommend for future work.

One final caveat: We intend this document to provide a road map of the GCL. We try to convey the design philosophy and the intuition behind the structures and classes making up the GCL. This document is not necessarily accurate in every detail. Indeed the limited budget forces this document to be incomplete. The ultimate reference is the source code, in which there are no known bugs.

1.4. Credits

I thank David Kaplan, our project leader, for his vision, foresight, and perseverance in conceiving of the basic ideas in PGM, for providing guidance and support during the development of the PGM Tool (PGMT), and for insisting that we implement the features described in PGM, compromising only when necessary.

This project was a multi-year effort involving several team members who devoted countless hours to put it together. My thanks go to all of the team members who worked on this project in the past.

I thank Wendell Anderson for frequent discussions, brilliant insight, and creative ideas that helped me in the design of the GCL.

I thank David Armoza, who helped with writing the code and testing in the early stages of developing the GCL.

Finally, I thank Ali Boroujerdi, who helped in the later stages with the implementation of the reassignment algorithm.

2. Families

We assume the reader is familiar with the notion of a family as described in the PGM Spec [1]. In PGMT, we strove to implement families in such a way as to conserve memory while promoting throughput and supporting the formatting of data to conform to the requirements of message passing via MPI. To do this, we recognized that the GCL uses families in two different ways:

- A dynamic family structure that changes over time during the life of the family, for example the family of tokens stored in a queue.
- A static family structure that does not changed during the life of the family, for example, a family of nodes, a family of included graphs, or a data token.

We observed that the dynamic family structure occurs in PGM only in the context of a sequence of tokens, where tokens are stored first-in-first-out, and on rare occasions deleted from the middle. To implement the storage of tokens in a queue, we chose to use the deque class, defined in the STL.

Using a deque to store static families requires much memory. Moreover, accessing an element by index is awkward and slow, particularly if the family height is large, requiring storage in the form of a deque of deques of deques, etc. There being no container class in the STL that is well suited for the static family structure, we designed and created our own. To implement the family tree, as described in the PGM Spec [1], we defined a special class called the *descriptor*.

Before we discuss the design and structure of the descriptor, we note that dynamic addition and removal of elements from this static family structure is awkward and slow. Thus we provide automatic conversion between the static and dynamic forms when it is appropriate to do so. This conversion is transparent to the user.

2.1. Descriptors

In this discussion we distinguish between a family with height 0 and a family with positive height. As noted in the PGM Spec [1], a family with height 0 is equivalent to a single element, or leaf. To unify the management of elements in PGMT, we consider that every element belongs to a family. If the family has height 0, in which case there is a single element, we say that it is a *trivial family*. Conversely, a family with positive height is a *non-trivial family*.

Storage of a static family has two parts: the *data vector* and the *descriptor*. The data vector stores all the leaves of the family from left to right depth-first. This is equivalent to the lexicographic order induced by the indexes. We accomplish direct access to the data vector with a 0-based single index.

We define the descriptor as a class containing a vector of integers that provides sufficient information to construct the family tree. In addition to methods for converting between the dynamic and static forms, we provide methods in the descriptor class to convert between the family index vector the data vector index for any given leaf in the family. Without confusion, we refer to the *descriptor* and the *vector of integers* synonymously.

The descriptor has a recursive structure. In all descriptors, the first two integers give the family height and the number of leaves. These two integers make up the descriptor's *preamble*.

If the family is trivial, then the descriptor has only the preamble. The height is 0, and the number of leaves is 1.

The default form of a non-trivial descriptor contains a concatenation of the descriptors of its children, without their preambles, because the height and number of leaves for each child can be calculated.

If all the children's descriptors are the same (i.e., if they have identical family trees), we say that the family is *regular* and that its descriptor is *regular*. Otherwise the family and the descriptor are *irregular*. A regular descriptor contains one copy of the common descriptor of all the children. One additional integer is provided in the header to give the *stride*, which is the number of leaves in each child. This results in a significant reduction in the size of the descriptor. There is also some reduction in the processing time of some of the methods.

Note that a regular family with height 2 is equivalent to a 2-dimensional array. Moreover the data for such a family is stored in the data vector in the same order as in a 2-dimensional array in the C language. This enables us to pass an array to a third-party routine without having to copy the data. We anticipate that in the majority of cases, families will be regular, and so the special treatment for regular families will result in a significant reduction of memory use.

Following the preamble is the header, which has between 2 and 4 integers. The first two integers of the header are present in every descriptor with non-zero height, and the third and fourth integers are optional:

1) Status word whose bits indicate information about the form of the descriptor, e.g., whether or not the descriptor is regular, and other information that is used to determine whether the 3rd and 4th integers of the header are present.
2) The number of children.
3) The lower bound of indices for the children (present only if non-zero). There is a bit in the status word to indicate whether the lower bound is zero.
4) The *stride*. The stride is the common number of leaves in each child of a regular family. It is present only for non-empty regular families with height > 1. Note that a family with height 1 is necessarily regular, and each child has height 1, implying that its stride is 1. There is a bit in the status word to indicate that the stride is present in the descriptor.

In an irregular descriptor, recall that not all children have the same descriptor. In this case we concatenate all of the descriptors of the children. After the header and before the concatenated descriptors of the children, there is a sequence of pairs of integers, one pair for each child. The

first of each pair is the relative index for the start of the child's data in the data vector, and the second of each pair is the relative index for the start of the header of the child's descriptor. The concatenated descriptors follow this sequence of pairs.

3. Graph Construction

We designed the graph structures to meet the following goals:

- Identification of families of nodes and included graphs by their user-assigned family names and indices, including the hierarchy of included graphs.
- Identification of node ports, graph ports, and included graph ports by their user-assigned family names and indices.
- Fast execution by *flattening* the graph. This means resolving the port connections between nodes in different included graphs.
- Support of user-defined data types as base types for tokens.
- Construction of a copy of a graph or included graph. Note: This is not a requirement, and we have not implemented copy construction. Nonetheless, the structures for graph construction are designed so that copy construction can be implemented relatively easily.
- Support of the various interface documents [4, 5, 6].

These goals led us to separate the structures of graph construction from those of graph execution. The structures of graph construction are concerned with families of nodes, families of included graphs, and families of ports, as specified by the user.

We defined a general class called a *graph object*. Subclasses of graph objects are classes of *main graphs, nodes*, *included graphs*, *node ports*, and *main graph ports*. (Note that we do not define included graph ports as graph objects. We will discuss this point in more detail below.) For each family of graph objects, we construct an object called a *handle*, and, accordingly, we define a class of handles.

Each main graph and each included graph contains a handle table, which is implemented as a vector of pointers to the handles. The following diagram shows the relations between the handle table, the handles, and the objects contained in each handle. We show only the significant objects; we encourage the reader to examine the code to learn the details.

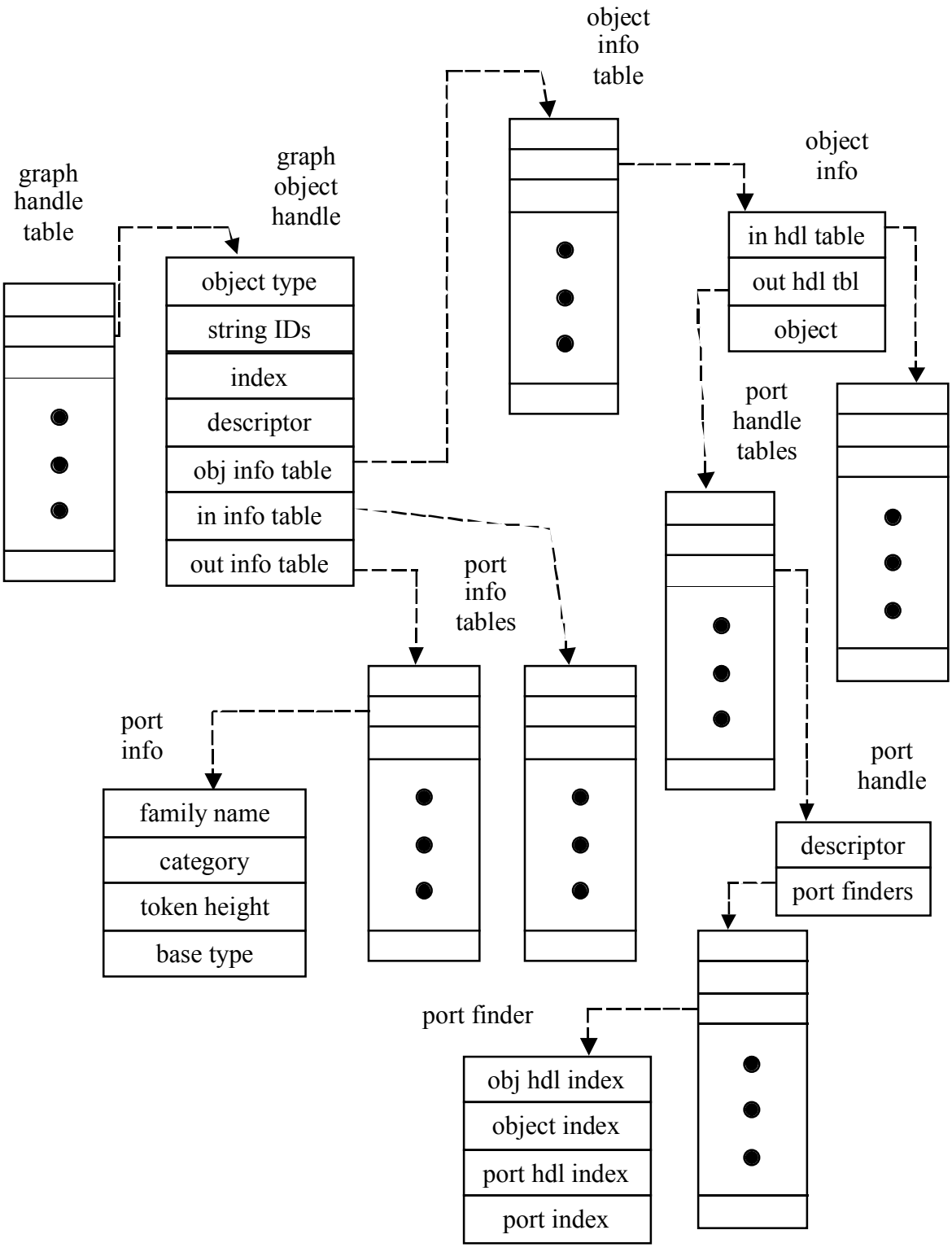


Fig. 3.1

In the above diagram we use English phrases to identify the various objects. In the GCL, the class names all begin with GCL_. The field names are either private or protected and, by

convention, have names that begin with an underscore. Aside from that, the class names and field names in the GCL are similar to those in the diagram above. For example, the graph object handle class name is GCL_graphObjHdl, and it is defined in the file GCL_graphObjHdl.h and GCL_graphObjHdl.cpp. The respective field names in this class are called _graphObjType, _stringIDs, _index, _descriptor, _graphObjInfo, _inportInfoVector, and _outportInfoVector. Note that some objects in the above diagram that are called tables are implemented as vectors. Accordingly, their field names are called vectors, rather than tables.

Unless otherwise stated, all classes are defined in files with same name and extensions .h and .cpp.

The graph handle table is an object of the graph (main graph or included graph). Each handle, an object of the class GCL_graphObjHdl, contains the following information about its family of graph objects:

- object type (_graphObjType) is a value of the enumerated type GCL_GraphObjType, defined in GCL_Types.h. This is the common type of the elements in the family.
- string IDs (_stringIDs) is a pointer to an object of the class GCL_StringIDs. This contains a number of strings associated with the class definition. The family name is one of the strings.
- index (_index) is an unsigned integer that gives the index of the handle in the handle table.
- descriptor (_descriptor) is a pointer to the family descriptor.
- obj info table (_graphObjInfo), labeled "object info table" in the diagram, is a vector of pointers to objects of type GCL_GraphObjInfo, labeled "object info" in the diagram. The graph objects in this vector are the leaves in the family. Accordingly, the size of this table is equal to the number of leaves specified by the descriptor. Fields in the object info are
  - object (_graphObj) is a pointer to the graph object.
  - in hdl table (_inportHdlVector) and out hdl tbl (_outportHdlVector) are vectors of pointers to objects of type GCL_PortHdl, labeled "port handle" in the diagram. These are the port handles of the graph object. Each port handle has the following fields:
    - pointer to a descriptor (GCL_Descriptor) is the descriptor of the port family
    - vector of port finders (GCL_PortFinder). Each port finder corresponds to a specific port (input or output, depending on which vector contains it) of the graph object. Note that four indices are needed to identify the port (assuming the direction of the port is known - input or output):
      - obj hdl index (identifies the family of graph objects in the graph),
      - object index (identifies the family member in the family,
      - port hdl index (identifies the family of ports in the graph object),
      - port index (identifies the port in the port family).

      We use the port finder when making a connection between node ports. Each port finder provides the information sufficient to identify the port to which it is connected. If the handle object type is "included graph", then we must resolve the port finder to a node port in that included graph. To do this, we refer to the handle table of the included graph to get the handle of the respective included graph port. Obtaining the port finder for the port in the included graph, we do a recursive search for the node port.
- in info table (_inportInfoVector) and out info table (_outportInfoVector) have the same form. Each is a vector of information (labeled "port info" in the diagram) that is common to

the port structure of every element in the family. The size of each port info table matches the size of the respective port handle table, and the identity map defines the correspondence between the port info and the port handle. Each port info contains the following information:

- family name (_familyName of type string) specifies the port family name.
- category (_category of type GCL_PortCategory, enumerated type defined in GCL_Types.h) specifies that the port is either a transition port or a place port.
- token height (_tokenHt of type unsigned int) is the token height of tokens passed via the ports in the port family.
- base type (_baseType of type GCL_BaseType, defined in global.h) is the common base type of the tokens passed via the ports in the port family. Recall that the mode consists of the token height and base type. In the GCL, during connection of two ports, we use the mode to verify compatibility of the two ports.

  Note that the base type may be a primitive language type (e.g., int or float) or it may be user-defined. The type GCL_BaseType is not really the base type, but a variable that identifies the base type. MTOOL, defined within the Middleware module, provides this variable by a function call with an argument that is a string identifying the type. For more information about MTOOL and the definition of user-defined types, see the appropriate documentation for the Middleware.

The next diagram shows the class hierarchy of the GCL classes involved in graph construction.

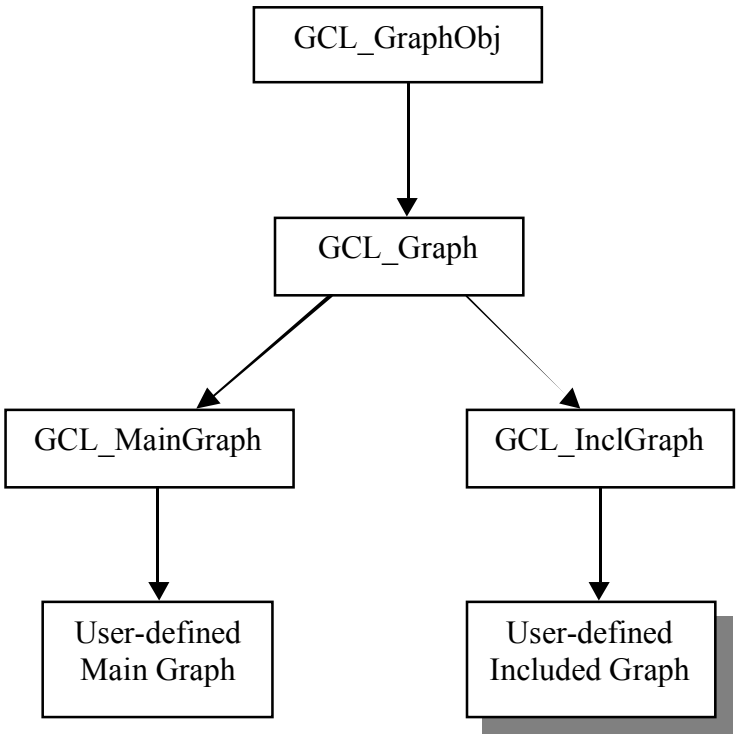


Fig. 3.2

Each arrow points from the base class to the derived class. All other classes involved in graph construction stand alone, and there is no hierarchy. We expect that the user will define the Main Graph and Included Graphs by means of the GUI and the translator. All other classes in this diagram are in the GCL.

Also derived from the class GCL_GraphObj are classes involved in graph execution. We will address those classes in the next section.

A complete discussion of graph construction necessarily includes a complete description of every class and method for graph construction. These descriptions are given in Appendix A.

During the construction of the handles to represent families of graph objects, we also construct the individual nodes of the graph and make the direct connections between the respective node ports. In doing this, we resolve all family indexing, so that each node has its own identity. Moreover, we resolve all of the included graph structure, so that intermediate ports of included graphs are bypassed. In this way we make direct connections between respective node ports, providing improved performance during graph execution.

4. Graph Execution

Fig. 4.1 shows the class hierarchy of graph objects involved in graph execution:

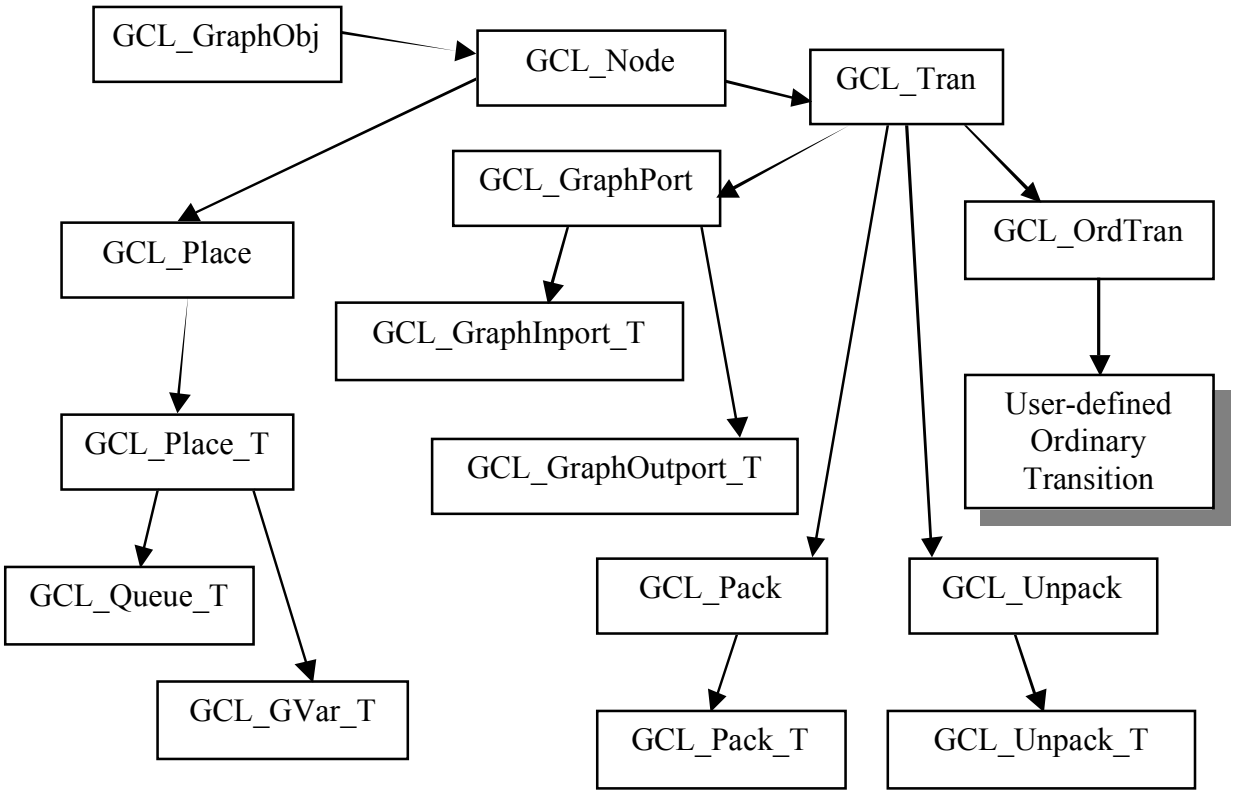


Fig. 4.1

As in the case of user-defined main graphs and included graphs, the GUI supports user definition of ordinary transitions. All other classes in this diagram are defined in the GCL.

Classes with names ending in _T define class templates. In each case the template argument is the base type of the tokens being stored or processed.

We had problems compiling class templates when we had separate header files and body files (with extensions .h and .cpp respectively). Thus we followed the lead of the Standard Template Library and defined all the methods of the class templates in the header files, thereby avoiding the body files. While this solved the problem, it created an inconvenience. The main program became quite large, and thus somewhat time-consuming to compile.

To minimize the effect of longer compile times, we strove to define methods in class templates as little as possible. We did this by defining respective base classes that are not template-specific, in which we defined many of the methods. We referred to such base classes as *wrapper classes* and derived the class templates from the wrapper classes. This technique also allowed us to define methods in the wrapper classes required by the GCL/PEP Interface Document [5].

The class GCL_GraphPort is derived from GCL_Tran. This is a special case of transition that is hidden from the user. Unlike all other transitions, graph ports execute upon request from the command program, either to produce data to a place in the graph or to read data from a place in the graph.

All other transitions (user-defined ordinary transitions and the special transitions pack and unpack) execute according to the rules spelled out in the PGM Spec [1]. We now discuss the execution of these transitions.

The PEP has the responsibility to keep track of the transitions that are ready to execute. The GCL has the responsibility to determine when a transition is ready and report that readiness to the PEP. Conversely, when a transition that is ready becomes not ready, that change must also be reported to the PEP. We designed the GCL according to the following ideas:

- Initially (i.e., during graph construction) all places (queues and graph variables) are empty. One of the last events during graph construction is to initialize the places with the respective tokens that were specified by the user.
- As the state of each place changes (i.e., as tokens are either produced or consumed - either during initialization or during graph execution), that place must notify all upstream and downstream transitions of the state change.
- Upon notification of a change of state of an upstream or downstream place, each transition analyzes its readiness to execute. If its state changes from "not ready" to "ready" or vice versa, that transition notifies the PEP, which then adds or removes the transition from its ready queue, as appropriate.
- When the PEP decides that a transition should execute, it calls the transition's execute method. The execute method causes the following to occur:
    - Input tokens are read from the upstream places.
    - The transition statement (a transition method) executes.
    - Output tokens are produced to the downstream places.

- Tokens are consumed from the upstream places.

To aid in the determination of a transition's readiness, we define a class called GCL_PredTable. Each transition contains a predicate table. The class hierarchy of predicate tables appears in the next diagram:

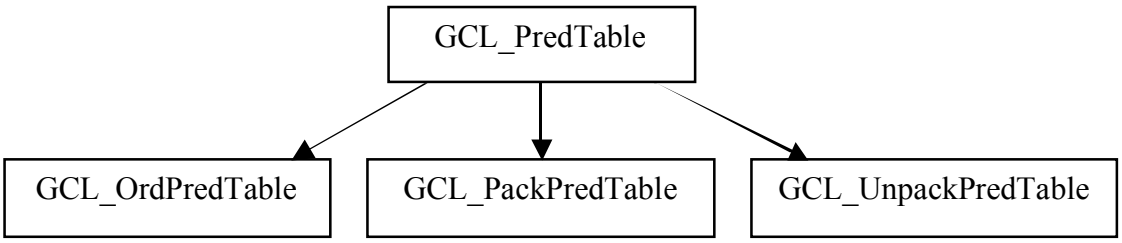


Fig. 4.2

Each predicate table contains two predicates in addition to a predicate for each input port and each output port. The first two predicates respectively indicate whether the transition is activated (i.e., unsuspended) and assigned to the current process. We discuss process assignment in the next section.

The predicate corresponding to an input port is true if and only if the respective upstream place meets its threshold (i.e., has enough tokens). The predicate corresponding to an output port is true if and only if the respective downstream place has sufficient available capacity to accept tokens to be produced.

The transition is ready to execute if and only if all predicates are true.

The predicate table for ordinary transitions is generic and is designed to accommodate every ordinary transition. The predicate tables for pack and unpack are specially written, because of the special requirements for readiness of pack and unpack. In particular, the PGM Spec [1] stipulates that the pack can be ready even if some or all of its NEP ports are not connected, in which case default values are assigned.

Every token is an instance of the class template GCL_Token_T, which is derived from the base class GCL_Token. There are token methods to support access to the token's descriptor and data, as well as to support construction of tokens. To aid the user in writing transition statements, there is a class template GCL_Workspace_T, which is derived from the base class GCL_Workspace. Every transition port family contains a workspace. For each input port family, the associated workspace provides access to the input token. For each output port family, the associated workspace supports construction of the output token.

The transition statement receives a single token from each input port family. The GCL automatically assembles this token from tokens read at all the ports in the input port family. Similarly, the transition statement must create a single token for each output port family. The GCL automatically disassembles this token to provide an output token for each port in the output port family.

Design notes for the Pack Transition and its predicate table appear in Appendix B. The design for the Unpack Transition is similar to that of the Pack Transition.

5. Distributed Processing

As noted above, we chose to use MPI for message passing between processes. While it is not required, it is very convenient to implement applications with MPI by *single program/multiple data*. This means that the entire application is loaded into each process; the processes run the application asynchronously and concurrently. By calling Middleware functions, which in turn call MPI functions, the application program can determine what the current process is and execute code that is predicated on that information. We chose to implement PGMT in this way.

Thus every node and every place has an instance in every process. Moreover, we do not have to be concerned with sending code via messages between processes.

For the above reasons as well as to minimize the thrashing that could result from dynamically assigning each transition to a process each time it executes, we follow these rules:

- The PEP makes the decision about which transitions are initially assigned to each process. The GCL is responsible to implement the PEP's assignment decision.
- Each transition is initially assigned to a unique process. In the absence of reassignment initiated by the PEP, every execution of the transition will occur on its assigned process.
- Each place (i.e., queue or graph variable) is assigned to the same process as its downstream transition(s).
    - The main advantage of this is that every token produced to a place is immediately sent by message to the process where it will be stored while waiting for the next transition to process it.
    - The disadvantage is that all transitions downstream from the same place must be assigned to the same process and therefore must execute sequentially. We felt that this is a small price to pay, because it would be rare for the user to specify multiple transitions downstream from a common place.
- During graph execution, it is necessary for messages to be sent between processes. Each message is sent from the instance of a place in one process to the instance of the same place in another process. There is one exception to this rule, which we explain in the next section on reassignment.
- The PEP decides when to execute each transition. The GCL is responsible to execute the transition.
- While the graph is executing, the PEP monitors the processing to determine whether the transitions are assigned in a way that meets the performance requirements. As a result, the PEP may decide that the transitions must be reassigned to improve performance. Having made this decision, the PEP initiates a reassignment by notifying the graph. The GCL is responsible for implementing the reassignment, and will do so with little if any disruption in the graph processing.
- The PEP shall not initiate another reassignment until after the previous reassignment is complete. This implies that the GCL must notify the PEP upon completion of the reassignment.

Some of the processes are dedicated to running the command program. We call these the *command program processes*. The remaining processes are dedicated to graph processing, and we call these the *graph processes*.

The PEP may reserve one or more of the graph processes for special use by the PEP. These special use graph processes execute what we call the *über-PEP* and are not directly involved in the execution of transitions. The über-PEP has the responsibility for creating the initial assignment, for monitoring performance, and for creating reassignments.

Each graph process not running the über-PEP is available for assignment of transitions and for executing the graph. Each such process has a *local PEP*, which keeps track of the readiness of locally assigned transitions and for initiating their execution. We call these processes the *local PEP processes*. Each local PEP also communicates with the über-PEP.

In each local PEP process, the local PEP controls the processing by executing a quasi-infinite loop. This loop will terminate upon the occurrence of an external event that signals operator intent to suspend or terminate graph processing. The PEP loop executes the following algorithm:

- Receive all incoming GCL messages. The PEP calls the graph method GCL_Graph::receiveMessages(), which executes a loop to receive and process all incoming GCL messages. Except as noted in the subsection below on reassignment, each such message was sent from the instance of a place in another process, addressed to the instance of the same place in the local process.
- Execute a transition by calling its execute method. The PEP selects a transition from its priority queue in which it keeps track of ready transitions in the local process.
- Perform internal PEP functions.

The command program assigns each graph port to one of the command program processes and passes this information as an argument to the user's graph constructor. Other arguments to the user's graph constructor include a vector of the command program processes, a vector of the graph processes, some strings to identify the graph and the version, a latency constraint for the PEP to use in determining assignment of transitions to processes, and GIPs (Graph Instantiation Parameters), which are defined in the user's graph..

During graph construction, the user's graph constructor calls the main graph method GCL_MainGraph::_initializeGraph. The user's graph constructor must be called in every graph process and in every command program process to which a graph port is assigned.

The translator generates the code for the user's graph constructor, which calls _initializeGraph after all the nodes and included graphs have been created. This method performs a number of necessary functions, among which are a call to the PEP method PEP::createAssignment, passing information that the PEP needs to determine the initial assignment of transitions to processes. After calling this PEP method, the _initializeGraph method implements the assignment that the PEP specified. Then, after every place in the graph is initialized in the processes where it is assigned, graph processing begins with the initial assignment of transitions to processes.

To initiate reassignment, the local PEP calls the graph method GCL_Graph::startReassignment in every local PEP process. When this method returns control to the PEP, reassignment has begun. Graph execution continues as described above (by PEP direction) during reassignment. Reassignment is complete when all local PEPs have been notified that reassignment is complete. The PEP is not required to synchronize the initiation of reassignment in all processes. We say that *reassignment starts* when the first local PEP initiates reassignment. We say that *reassignment ends* when the last local PEP is notified that reassignment is complete. From the time that reassignment starts until reassignment ends, we say that *reassignment is in progress.* If no reassignment is in progress, then we say that the assignment is *static*.

5.1. Distributed Processing During Static Assignment

In this section we discuss distributed processing while no reassignment is occurring. We discuss reassignment in the next section.

To implement reassignment, each node (i.e., each transition and each place) has a variable that indicates the process to which it is assigned.

As noted above, there is a predicate in the predicate table of each transition that indicates whether the transition is assigned to the current process. To encapsulate transition execution, we designed the transitions to be ignorant of whether its downstream place(s) are assigned to the current process. This means that the transition produces output tokens by calling the respective place's produce method.

It is thus incumbent on the place to determine whether the token(s) produced should be stored in the local process or sent by message to another process. In this discussion, we introduce the following notation about the instance of a node in each process. By the *current instance of a node*, we mean *the instance of the node in the current process*.

- If a node is assigned to the current process, we say that it is *locally assigned*.
- If a place is not locally assigned but has an upstream transition that is locally assigned, we say that the current instance of the place is an *agent*.
- Otherwise the current instance of the place has no upstream or downstream transitions that are locally assigned. In this case we say that its current instance is *idle*.

To accomplish distributed processing we implement the following design for queues. We use a similar design for graph variables with some of the steps eliminated, particularly in the case of consuming tokens, which is a null operation.

- If a token is produced to a queue that is locally assigned,
  - Store the token.
  - Update the content and available capacity in the local instance.
  - Notify all locally assigned upstream and downstream transitions.
  - Send messages to all agents with the change in available capacity.
- If a token is produced to an agent,
  - Send the token to the process where the queue is assigned.
  - Update the content and available capacity in the local instance.
  - Notify all locally assigned upstream transitions.
- If a "token" message arrives (i.e., one containing a token) for a locally assigned queue,

    - Store the token.
    - Update the content and available capacity in the local instance.
    - Notify all locally assigned upstream and downstream transitions.
    - Send messages to all agents other than the sender of the message with the change in available capacity.
- If a content update message arrives for an agent,
    - Update the content and available capacity.
    - Notify all locally assigned upstream transitions.
- If token(s) are consumed by execution of a (locally assigned) downstream transition,
    - Update the content and available capacity.
    - Notify all locally upstream and downstream transitions.
    - Send messages to all agents with the change in available capacity.

5.2. Reassignment

We designed reassignment with the goal to continue processing with little if any disruption. To do this, we added some message types to be used for reassignment. We did this almost completely within the rules mentioned above to the extent of having every GCL message originate from the instance of a place in one process, addressed to the instance of the same place in another process.

There is one exception to this rule, necessitated by the need for some command program processes to participate in the reassignment process. Note from the discussion above that each command program process is controlled by the command program and not by a local PEP. Nonetheless, if a command program process has an assigned graph port, then that process must be notified of the reassignment. To solve this problem, we altered the method GCL_Graph::startReassignment to have an argument that is a vector containing the local PEP processes. One of the local PEP processes is then appointed as the *master*. The local graph in the master process then sends a message to the graph in each command program process with an assigned graph port, notifying it of the reassignment. When the reassignment is complete in each command program progress, the respective graph sends a message to the graph in the master process. For the graph in the master process to notify its local PEP that reassignment is complete, in addition to waiting for notification by all its local node instances, it must wait for all the completion messages from the command program processes.

In the rest of this section we discuss the reassignment process for places. Owing to the complexity of reassignment, we first discuss an example to illustrate how reassignment works. We will then briefly discuss some variations of the example and indicate adjustments to be made for the different cases. In Appendix C we give the full design for reassignment.

Consider a graph with two transitions T1 and T2 that are connected by a queue Q. The entire graph may have a large number of transitions and places. For a place to accomplish its part of reassignment, the actions of that place are independent the reassignment of any other place in the graph.

For illustrative purposes we concentrate on this small part of the graph. Every place in the graph may be considered in a similar way. T1 and T2, initially assigned to respective processes 1 and 2, are reassigned to respective processes 3 and 4, as illustrated in Figure 5.1.

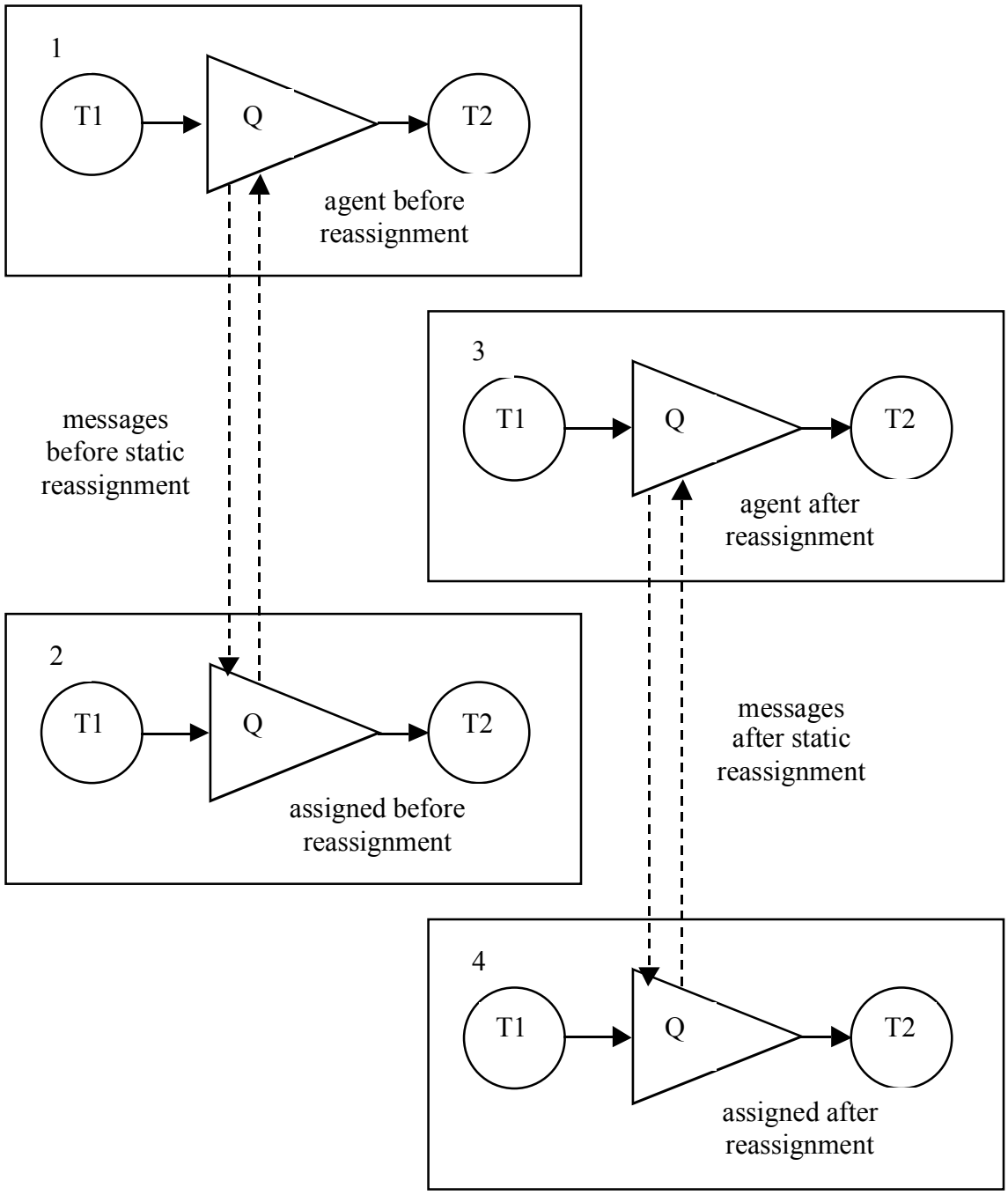


Fig 5.1

The dotted arrows show messages between the instances of Q during static assignment: between Q(1) and Q(3) before reassignment and between Q(2) and Q(4) after reassignment. In particular,

before reassignment, the messages from Q(1) to Q(2) contain tokens produced by T1(1), and the messages from Q(2) to Q(1) contain messages that tell when tokens are consumed during the execution of T2(2). After reassignment, the same communications will occur between processes 3 and 4.

We use the following notation: If N is a node and P is a process, we indicate the instance of N in process P by N(P). Thus, for example, the instance of T1 in process 3 is T1(3), and the instance of Q in process 2 is Q(2).

During reassignment, we must ensure that the proper order of tokens stored in Q is maintained as Q is reassigned from process 2 to process 4. Specifically, we must ensure that the tokens previously produced by T1(1) and stored in Q(2) are transmitted and stored in Q(4) before T1(3) starts executing and sending tokens to Q(4).

In the following table, each column indicates the sequence of events that must occur in the respective process. Except as noted, the events in different processes are asynchronous.

Q(1) [agent becomes idle]	Q(2) [assgnd becomes idle]	Q(3) [idle becomes agent]	Q(4) [idle becomes assgnd]
Block execution of T1(1).	Send a "done" message to Q(1) indicating no more "consume" messages.	Wait for "done" message from Q(4).	While waiting for "done" message from Q(2), store tokens arriving from Q(2), along with queue capacity. Notify T2(4), so it may become ready for execution.
Send a "done" message to Q(2), indicating no more token messages.	Send all currently stored tokens to Q(4). Include the current queue capacity.	When "done" message arrives from Q(4), Q(3) reassignment is complete.	When the "done" message arrives, send a "done" message to Q(4). Include content and capacity.
While waiting for "done" message from Q(2), ignore incoming "consume" messages.	While waiting for "done" message from Q(1), keep sending tokens to Q(4) as they arrive from Q(1).		Q(4) reassignment is complete.
When "done" message arrives from Q(2), Q(1) reassignment is complete.	When "done" message from Q(1) arrives, send a "done" message to Q(4) indicating that no more tokens will be forwarded.		
	Q(2) reassignment is complete.		

Fig 5.2

Note that the sequence in each column is synchronized with the other columns only by the indicated messages. The design accounts for the fact that the different column sequences may start at different times and proceed at different rates. For example, the "done" message from Q(1) may arrive at Q(2) before the reassignment sequence in Q(2) starts. In fact, in this example, any "done" message may arrive at its destination before its addressee is expecting it.

Note also that it is possible for the PEP to decide that T1 should remain assigned to process 1 while reassigning T2 from process 2 to process 4. Similarly, T1 may be reassigned to process 3 while T2 remains assigned to process 2. Or T1 may be reassigned from process 1 to 2 while T2 is reassigned from process 2 to process 4. The design must account for all possible variations.

Still another dimension of the complexity arises from the fact that a queue may have a non-trivial family of input ports. Then it will have multiple upstream transitions, possibly assigned to different processes, implying multiple agents. If a queue has multiple downstream transitions, all the downstream transitions must be assigned to the same process, and so the complexity to account for this case is limited.

Reassignment for a graph variable is similar to, but a little simpler than, reassignment for a queue. We do discuss reassignment of graph variables in this document.

To implement reassignment, we designed each place as a finite state machine. During static execution, each instance of place is in the *initial* state playing one of three roles: assigned, agent, or idle. The goal is to transform that instance to the reassigned role (assigned, agent, or idle) by passing through a sequence of states as the various events occur. The receipt of a reassignment "done" message from another process triggers each such event.

When an event occurs (either before or after the instance starts waiting for it), a flag is set to indicate that the event has occurred, and the GCL_Place::reassign() method is called to process the event.

On its first call, the reassign method determines the necessary sequence of states for the place to accomplish its reassignment. Each state has an associated event, which must occur before the place can move to the next state. Following the establishment of the state sequence, the reassign method enters a loop. In each pass through the loop, the event flag for the current state is checked. If the event has occurred, the appropriate actions are taken, the place moves to the next state in the sequence, and the loop is reentered to check for the next event. If the event has not occurred, the reassign method terminates, ultimately returning control to the PEP for further processing. At some time later, when an event occurs, the reassign method is again called, and the loop is reentered in the current state, and the process is repeated. When the place reaches the final state, completing its reassignment, the reassign() method resets all the event flags, moves the place to the initial state, and reports to the graph that the place has completed reassignment.

6. Testing & Validation

We wrote the test software as we went along. There are two groups of tests.

The first group of tests was set up to test the descriptor and token classes. These tests are in the cvs repository, in the module GCLGram. Compiling these tests may require modification of one or more makefiles to account for the recent change of the filename GCL_Descriptor.C to GCL_Descriptor.cpp.

The second group of tests assumes that the descriptor and token classes have been tested and validated. These tests were designed for all of the GCL classes other than the descriptor and token classes. They are not entered into the cvs repository. Rather they are stored in the compressed tar file /home/stevens/pgmt/sandbox/archive99.tar.Z.

We say a few words about this test environment. Upon uncompressing and expanding this file, one should change to the directory sandbox/Building. In the rest of this section, we assume this is the current directory.

There is a readme file that we intended to explain all the tests and how to run them. This is one of many items of unfinished business. The readme file currently lists those tests that run on multiple processes, and gives the number of processes required for each. Each of the other tests runs on a single process.

Before compiling any of the tests one should execute

%gmake depend

This will cause the Makefiles in the current directory and subdirectories to append the dependencies. One may also remove all compiled files (i.e., ones with extension .o and executable files) by executing

%gmake clean

Finally, one may remove all the dependencies from the Makefiles by executing

%gmake undepend

There are about 55 tests, which collectively make up the regression test suite for the GCL. As an example, to compile and run test2, execute the following command lines:

%gmake LOCAL_GCL=1 test2
%GCLGram_F

The following three variables can be set to affect the code that is compiled:

DEBUG_GCL to cause debug information to be printed to a file pgmt0.log, pgmt1.log,…, where the integer is the process identifier. There will be a separate log file for each process. We

recommend using this feature sparingly and selectively, as the resulting I/O can drastically slow the processing, and the output file(s) can be very large (in some cases tens or hundreds of megabytes). As a result, the information in the files will be too voluminous to wade through in a reasonable way.

LOCAL_GCL to indicate that the application will run on a single process. This disables a test to make sure that the command program processes and PEP processes are in disjoint sets.

PURE_GCL to indicate that the test will be compiled and run with Rational Purify. Purify has problems when exceptions are thrown. This avoids the problem by disabling some code that will throw exceptions. This is set automatically when compiling to run Purify. By way of explanation, we developed GCL with zero tolerance for memory leaks, and accordingly we made extensive use of Purify in the testing process.

All of these variables are set to 0 by default in Makefile.incl. Any combination of these variables may be set in the compile command line.

For example, to compile and run test2 with Purify, execute the commands
%gmake LOCAL_GCL=1 pure2
%GCLGram_F.pure >& pure.log

In the compile command, the variable LOCAL_GCL is set for the reason described above. Note that the variable PURE_GCL is set automatically by the Makefile, and so it is not necessary to do so explicitly. In the run command, all output is redirected to the file pure.log, where it can be examined after the run. This file is likely to be quite large, and failing to redirect will cause an impossibly large amount of data to be written to standard out (i.e., to the user's screen).

To compile and run a test that runs on distributed processes, we cite test23 as an example. We refer to the readme file. We assume that the user is logged in on Sun machine named "shiloh", running under the solaris operating system:

%gmake test23
%mpirun -np 3 -machinefile allshiloh Build4_int

In general, the name of the executable file for each test can be found by looking in the Makefile under the section for the respective test.

Finding a bug by setting the variable DEBUG_GCL in the compile command is not unlike looking for the proverbial needle in a haystack. An alternative approach, which we found to be workable, is to edit the source files by commenting out the lines that prevent the diagnostic code from being compiled, typically in code segments like the following:

```
#if DEBUG_GCL
// some diagnostic code normally not compiled
#endif DEBUG_GCL
```

7. Future Work

There are many items of unfinished business and of enhancement in the GCL, some of which we list here in no particular order:

- Complete this document.
- Complete in-line comments in the GCL code.
- Finish writing the readme file describing the test environment.
- Modify the test environment (in the compressed tar file) for inclusion in the cvs repository.
- Redefine the exceptions. At present the exceptions are too specific and too numerous. Moreover, they are not sufficiently informative to be particularly useful. With time and resources, we would revise the entire set of exceptions. In particular, they should be written to provide a stack trace of procedures called at the point where the exception occurred. They should also identify the object where the exception occurred, in terms the user can understand.
- Design and implement class templates for two transitions called *switch* and *merge*. Note that transitions performing these functions are not ordinary transitions, because they have ports that violate the rule exactly one token to be read and consumed via every input port and one token to be produced via every output port. It is possible to write included graphs that perform these functions, as we describe in Appendix D. Implemented as included graphs, performance suffers because of the copying of data. This copying can be largely eliminated if we wrote the transitions. We feel that these functions are sufficiently useful that the value of PGM will be enhanced by the inclusion of switch and merge transitions.
- Provide a means by which the user can specify that an input token in an ordinary transition should be passed on and produced via an output port. In the present GCL implementation, to do that requires making a copy of the token. If the token contains a very large data structure, and if the transition makes little or no change in that data structure, copying can be expensive, both in time and memory. In this case, performance can be significantly enhanced by allowing the user to specify that an input token should be produced without copying it. We note that with this feature, still further performance improvement can be realized with switch and merge.

8. References

1. *Processing Graph Method 2.1 Semantics*, by David J. Kaplan and Richard S. Stevens, April 1, 2001, hereinafter called the PGM Spec.
2. *Design Notes for GUI Forms*, by Dick Stevens, March 27, 2001. This document specifies the look and feel as well as the functionality of the GUI.
3. *The Graph State File Specification*, by Kalpana Bharadwaj, January 27, 2000. This document specifies the grammar for the Graph State File (GSF), to be used by the GUI for storing a graph being edited with the GUI.
4. *Specification for the Translator Output*, by Dick Stevens, March 27, 2001. This document specifies how the translator will generate code from a GSF.
5. *Some PGMT interfaces*, by Joe Collins, April 25, 2000.
6. *GCL/MW Interface*, by Roger Hillson, July 1. 1999.
7. *The Unified Modeling Language User Guide*, by Grady Booch, James Rumbaugh, Ivar Jacobson, Addison Wesley, 1999.

Appendix A
Classes and Methods

TBS

Appendix B
Design Notes for Pack Transition

Dick Stevens

Friday, August 18, 2000

NOTE (Saturday, April 21, 2001): I wrote these notes to correct problems in an earlier flawed implementation of the Pack Transition. I beg the reader's indulgence. With sufficient time, I would revise this appendix. Further, if the reader notices a discrepancy between these notes and the code in the GCL, the code is the final authority on the implementation.

I. Background

A. The Pack Transition, in its current implementation, fails to determine correctly when it is ready to execute. This document outlines the changes to be made.

B. The class PackPredTable will undergo a complete redesign, together with its interface with the classes Pack and Pack_T. In addition, we implement a new kind of transition port, which notifies the Pack of all events occurring in the upstream place. The reason is that the existing transition port class checks whether there is a change that affects the state with respect to threshold, preventing the Pack from being notified of some events.

II. Pack Transition - High Level Design

A. Special management of the Nep ports and the INPUT port are necessary. Unlike an ordinary transition, it is possible for the Pack transition to execute if some or all of its Nep ports are not connected. Accordingly, no warning is issued about unconnected Nep ports of a Pack transition. In the situation where the threshold = 0, it is even possible for the Pack to execute if its INPUT is not connected.

B. We now address the determination of a pack's readiness to execute. While the requirements for an ordinary transition to be ready can be met in any order, the order is important for a Pack Transition. This order must be observed even though the events may occur in any order. In particular, all of the Nep values must be known before the threshold for the INPUT can be calculated. Until the threshold is known, the INPUT port is, by definition, not ready. Note that special care must be taken to ensure that default values are properly calculated for Nep ports that are not connected. For unconnected Nep ports, the default values are determined as follows (as specified in the PGM 2.0 Specification):

   a. READAMOUNT = 1.
   b. READOFFSET = 0.
   c. CONSUMEAMOUNT = READAMOUNT.
   d. CONSUMEOFFSET = 0.

C. We assign to the Pack's predicate table all the responsibility for determining the Pack's readiness to execute. Because of this, many of the Pack's public methods will be implemented in the Pack's predicate table. To support this implementation, we give the

predicate table access to the Pack's ports. We note the tight coupling between the Pack and its predicate table.

D. Initially (i.e., during graph construction, after all nodes are constructed, and before the places are initialized), the predicates in the Pack's predicate table are set to indicate that no ports are connected, so that they will take their respective default values. Then all connected Nep ports are made not ready. We give the following algorithm for determining readiness of a Pack:

1. Pack not currently executing: For each event reported by a place, we proceed as follows. Each case a, b, or c below describes what to do for each respective event.
   a. Notification of available Capacity for OUTPUT:
      if (Available Capacity > 0) {set OUTPUT ready } else {set OUTPUT not ready }
   b. Notification of content for INPUT:
      if (all Neps ready) {
            calculate threshold from Neps
            if (content >= threshold) {INPUT ready}
            else {set INPUT not ready}
      }
      else {set INPUT not ready}
   c. Notification of content for a Nep:
      if (content > 0) {
            Set the Nep ready
            Read the Nep from the place (Note: this must be done even if it was
                  previously read, because it may be a new value of a graph variable.)
            if (all Neps known) {
                  calculate threshold from Neps
                  get INPUT content // Note: INPUT content = 0 if INPUT not connected
                  if (INPUT content >= threshold) {set INPUT ready}
                  else {set INPUT not ready}
            }
      }
      else {
            Nep not ready
            INPUT not ready
      }
      Note: in case c, if the INPUT is not connected to a place, then INPUT content = 0.

      Following each event, if the Pack was not previously ready and is now ready, then the PEP is notified to put it on the ready queue. If the Pack was previously ready and is now not ready, then the PEP is notified to take it off the ready queue. Otherwise the PEP is not notified.

2. Pack currently executing: The same algorithm is used as above, except as follows:

a. During production to OUTPUT and consumption from the Nep ports and from INPUT, notification by the respective places will occur, as described in I.A and I.B. above. These events are processed according to the algorithm described in II.C.1 above, *except* that no determination is made after each event about the overall readiness of the Pack for a subsequent execution.
b. After all production and consumption, a determination of the Pack's readiness is made. If the Pack is ready, then the PEP is so notified.

III. Pack Transition - Detailed Design

A. We eliminate the class NepPorts and incorporate many of the functions of the NepPorts class into the class PackPredTable. We also incorporate many of the Pack functions into the PackPredTable. We specify a complete new design for the PackPredTable. Where appropriate, we indicate changes in the interface between the classes PackPredTable and Pack. We also indicate changes in some of the algorithms of the class Pack. In general, we strive to reuse much of the existing code.

B. There is a new PEP interface for the Pack that requires the execute method to return a single unsigned int in the leaf count vector. That is the leaf count of all tokens read from the INPUT port. Accordingly, in the Pack constructor, we resize that vector to 1. The value is the same as the number of leaves in the output token and is inserted by the Pack's execute method after calling the transition statement.

C. The following Pack methods will directly call respective PackPredTable methods:
9. updateInput
10. updateOutput
11. initialize
12. initNeps

D. The Pack method updateInput(portID) shall do the following algorithm:

```
1. if ( not isExecuting ) { set wasReady = PackPredTable.isReady() }
2. PackPredTable.updateInput(portID)                                  // see M below
3. if ( not isExecuting ) {
        set nowReady = PackPredTable.isReady()
        if ( not wasReady and nowReady ) { notify PEP to put me on ready queue }
        if ( not nowReady ) {
            if ( wasReady ) { notify PEP to take me off ready queue }
            if ( threshold ready ) {
                if ( inport connected ) {
                    if ( input place capacity < 2 * threshold – 1 ) { notify PEP deadlock alert }
                }
            }
        }
    }
```

E. The Pack method updateOutput(outportReady) shall do the following algorithm:

1. if (not isExecuting) { set wasReady = PackPredTable.isReady() }
2. PackPredTable.updateOutput(outportReady) // see N below
3. if (not isExecuting) {
   set nowReady = PackPredTable.isReady()
   if (not wasReady and nowReady) { notify PEP to put me on ready queue }
   if (wasReady and not nowReady) { notify PEP to take me off ready queue }
   }

F. The Pack method execute() shall do the following algorithm:

- verify readiness of Pack to execute: if not ready, throw an exception
- set isExecuting = true
- get the threshold from the PackPredTable and set the last threshold
- get the consume amount from the PackPredTable and set the last consume amount
- readInputs() // see G below
- tranStmt // see H below
- enter the leaf count of the packed token into the 0 position of the return vector // see Note below
- produceOutputs () // see I below
- consumeInputs () // see J below
- if (packPredTable.isReady()) notify PEP to put Pack on Ready Queue // see S below
- else (not packPredTable.isReady()) { // added on Friday, August 18, 2000
  if (threshold ready) {
  if (inport connected) {
  if (input place capacity < 2 * threshold – 1) { notify PEP deadlock alert }
  }
  }
  }
- set isExecuting = false
- return the outputVector
  Note: The return vector definition has 1 unsigned integer, which tells the number of leaves read from the input port.

G. The Pack method readInputs() shall be defined in the Pack_T class and shall do the following algorithm:

- get the read amount and read offset from the PackPredTable
- read from the input place to get the token deque

H. The Pack method tranStmt() shall be defined in the Pack_T class do the following algorithm:

1. create the packed token from the deque of tokens
2. assign the packed token to the OUTPUT workspace
3. delete the TokenDeque

I. The Pack method produceOutputs() is the same as for an ordinary transition. We make no further comment here.

J. The Pack method consumeInputs() shall do the following algorithm:
    1. if the INPUT port is connected {
    a. get the upstream place from the INPUT port
    b. get the consumeAmount from the PackPredTable
    c. get the consumeOffset from the PackPredTable
    d. call the place method to consume the specified tokens
    2. for each Nep {
        if (the Nep port is connected) {
            have the respective workspace consume the token from the upstream place
        }
    }

    Note: It is very important that the above steps be executed in the stated order. Reason: The consumeAmount and consumeOffset must be evaluated for consumption from the INPUT port before consumption from the Nep ports. This clears the way for reading new Nep values upon consumption of the tokens in the places upstream from the Nep ports. See the updateInput method of the packPredTable.

K. For the PackPredTable, there shall be a vector of 5 structures, where each structure contains the following information about each Nep in the following order: threshold, readAmount, readOffset, consumeAmount, consumeOffset. The structure shall have the following fields:
1. unsigned int value
2. bool isConnected
3. bool isReady

L. The PackPredTable shall maintain the following data:
- _nepStatus (vector of 5 structs, as described above)
- _inportList (const vector<GCL_TranPort *> &) // alias of the Pack _inportList
- _outportList (const vector<GCL_TranPort *> &) // alias of the Pack _outportList
- _inputReady (bool, to indicate that the input is ready)
- _outputReady (bool, to indicate that the output is ready)

M. The PackPredTable shall have the following public methods to be called by the Pack:
- updateInput (args: unsigned int content) // see N below
- updateOutput (arg: bool portIsReady) // see O below
- initNeps() // to call _initNeps () // see R below
- isReady() // see S below
- initialize() // see T below

N. The PackPredTable method updateInput (args: portID) shall do the following algorithm:
    1. Verify that the portID is valid: if not, throw an exception
    2. set wasReady = _inputReady
    3. Get the content of the place upstream from the respective port.
    4. If (port is INPUT) {
    _updateInput (content) // see P below

}
5. Else (the port is a Nep) {
   _updateNep (portID, content) // see Q below
   }
6. _adjustForThreshold // see R below
7. set nowReady = _inputReady
8. if (not wasReady and nowReady) { increment _numReady }
9. if (wasReady and not nowReady) { decrement _numReady }

O. The PackPredTable method updateOutput(arg: bool portIsReady) shall do the following algorithm:

1. set wasReady = _outputReady
2. set _outputReady = portIsReady
3. set nowReady = _outputReady
4. if (not wasReady and nowReady) { increment _numReady }
5. if (wasReady and not nowReady) { decrement _numReady }

P. The PackPredTable method _updateInput (arg: unsigned int content) shall do the following algorithm:
   1. if (threshold is ready) {
      if (content >= threshold) { set _inputReady = true }
      }
   2. else (threshold not ready or content < threshold) { set _inputReady = false }

Q. The PackPredTable method _updateNep (args: NepID, content) shall do the following algorithm:
   1. if (content > 0) get the port and the workspace, read the token into the workspace, get the value of the token, and release the token from the workspace
   a. set Nep's value = token value
   b. set Nep's ready = true
   c. if (NepID == 1) { // i.e., readAmount
         if (consumeAmount not connected) set consumeAmount's value = token value
      }
   2. else (content == 0)
   a. set Nep's ready = false

R. The PackPredTable method _adjustForThreshold shall do the following algorithm:
   1. if (all neps are ready) {
      set thresholdReady = true
      set threshold = max(readAmount + readOffset, consumeAmount + consumeOffset)
      if (inputContent >= threshold) { set _inputReady = true }
      else (inputContent < threshold) { set _inputReady = false }
   }
   2. else (not all neps ready) {

set thresholdReady = false
set _inputReady = false
}

S. The PackPredTable method _isReady() shall do the following algorithm:
    1. return (_numReady == 4)

T. The PackPredTable method initialize(), to be called during graph construction at the time of transition assignment, before the places are initialized, shall do the following algorithm:
    1. for each nep {
    if (no place in corresponding port) {
    if (readAmount or consumeAmount) { set value = 1 } // default
    else (readOffset or consumeOffset) { set value = 0 }
    set connected = false
    set ready = true
    set constant = true
    if (consumeAmount) { set constant = readAmount.constant }
    }
    else (place in corresponding port) {
    set connected = true
    set constant = nepPlace.constant()
    }

U. The updateInput method of the TranPort class must be modified so that if the threshold is zero, whenever the method is called, the owner's updateInput method will be called unconditionally. Moreover, the pack constructor should specify a value of 0 for the threshold when it calls the TranInport_T constructor for the INPUT port and for each Nep port. This constitutes an overloaded meaning of the threshold in the TranPort class. However we deem that it will work and that it involves minimal programming effort.

V. Without going into details here, we will examine the data members and methods of the classes Pack and Pack_T with an eye toward moving them from Pack_T to Pack whenever possible.

W. Pack methods to get next consume and get next threshold – revise to call the corresponding methods of the PackPredTable.

Appendix C
Reassignment Matrix
Ali Boroujerdi and Dick Stevens
Thursday, June 29, 2000

The matrix on the next page outlines the algorithm for reassignment. All communication for reassignment within the GCL occurs between a place object Q in one process P1 and Q in a different process P2.

We explain some terminology that is used in the matrix:

4. Each place object Q in each process P plays a role. If Q is assigned to process P, then we say that the role of Q in P is *assigned*. If Q is not assigned to P and Q has an upstream transition assigned to P, then the role of Q in P is *agent*. If the role of Q in P is neither assigned nor agent, then the role of Q in P is *idle*.
5. During reassignment, a place object Q in P may change its role. We use the adjectives *old* and *new* to indicate the role before and after reassignment, respectively. For example, if Q in P is assigned before reassignment and agent after reassignment, we say that Q in P is *old assigned* and *new agent*.
6. During reassignment, synchronization may be needed between Q in process P1 and Q in process P2. Suppose, for example, that Q in P1 must wait for a message from Q in P2. We will indicate this by saying that Q in process P1 *waits for* "done" from P2 and that Q in P2 *sends* "done" to P1.
7. If Q in P is assigned and has multiple agents in processes other than P, Q in P may wait for "done" messages from some or all its agents, and P2 may send "done" messages to some or all its agents. In each case we will indicate which agents will await or send the "done" messages.

1. When Q is waiting for "done" messages it will be necessary to prevent locally assigned upstream transitions from executing. To *block* means that Q in P shall temporarily prevent all execution of local upstream transitions. When Q receives all the expected "done" message(s), Q in P will *unblock*, allowing the upstream transitions to execute.
2. When an old agent remains an agent, we can allow the upstream transitions to fire, holding the output tokens until it is OK to send them. To *hold* means we hold the output tokens. To *release* means we send the tokens on.

In the matrix we refer to the following notes:

4. If Q in P1 is new assigned and Q in P2 is old idle and new agent, then Q in P1 must send the capacity and content (C&C) to Q in P2. This is because an idle place is not continually aware of the capacity. NOTE: This fails because of a race condition. Kill it.
5. For the same reason, if Q in P1 is old assigned and Q in P2 is old idle and new assigned, then Q in P1 must send the capacity to Q in P2.

new / old	assigned	agent	idle

assigned	if (there is more than one upstream transition OR upstream transition is reassigned)
	• Block.
	• Send "content done" to old agents
	• Wait for "done" from old agents.
	• If (a queue)
	  • Send "done" with C&C to new agents (see note 1 above).
	• Unblock.
		• Block.
		• If (a queue)
		  • Send "content done" to old agents
		• If (no old agents pending)
		  • send "done" with token ID, capacity, and tokens to new assigned
		• Else
		  • Send token ID with tokens to new assigned.
		  • Wait for "done" from old agents.
		  • Send capacity and "done" to new assigned.
		• Wait for "done" with C&C from new assigned (see note 1 above).
		• Unblock.
			• Block.
			• If (a queue)
			  • Send "content done" to old agents
			• If (no old agents pending)
			  • send "done" with token ID, capacity and tokens to new assigned
			• Else
			  • Send token ID with tokens to new assigned.
			  • Wait for "done" from old agents.
			  • Send capacity and "done" to new assigned.

agent	• Block.
	• Send "done" to old assigned.
	1. Wait for token ID and "done" with capacity from old assigned (see note 2 above).
	• Send "done" with C&C to new agents (see note 1 above).
	• Unblock.
	• If (a queue)
	  • Wait for "content done" from old assigned.
		If (there is more than one upstream transition OR downstream transition(s) is (are) reassigned)
		• If only one upstream transition
		  • Hold.
		• Else
		  • Block
		• Send "done" to old assigned.
		• Wait for "done" with C&C from new assigned (see note 1 above).
		• If holding
		  • Release.
		• Else
		  • Unblock
		• If (a queue)
		  • Wait for "content done" from old assigned.
			8. Block.
			9. Send "done" to old assigned.
			• If (a queue)
			  • Wait for "content done" from old assigned.

idle	2. Wait for token ID and "done" with capacity from old assigned (see note 2 above). 3. Send "done" with C&C to new agents (see note 1 above). 4. Unblock.	4. Wait for "done" with C&C from new assigned (see note 1 above). 5. Unblock.	

Notes on implementing reassignment

To identify all the different message types implied by this design, we revise the list of message types. We group the message types according to whether they are intended for normal communication or for reassignment.

Following message types are intended for normal execution, but must be recognized and possibly handled differently if reassignment is ongoing:

```
TOKEN_MSG            // for sending ordinary token message outside of reassignment
CONTENT_MSG          // for updating agent's content - sent from assigned place
ASSIGNED_CAPACITY    // for updating agent's capacity - sent from assigned place
AGENT_CAPACITY       // for updating assigned place's capacity - sent from agent
```

Following message types are intended for use only during reassignment:

AGENT_DONE	// old agent notify "done" to old assigned
OLD_CONTENT	// old assigned notifies "done" to old agents
OLD_TOKENS	// old assigned forward token(s) to new assigned
OLD_DONE	// old assigned forward capacity and token(s) and notify "done" to new assigned
NEW_DONE	// new assigned notify new agents "done" with C&C

Ancillary functions and variables to support reassignment:

- When AGENT_DONE is received:
  - decrement the number of agents pending counter (it may go negative)
  - if reassignment has been started, call the reassign method

- When OLD_TOKENS is received:
  - store the current token ID (if contained in the message)
  - store the tokens

- When OLD_DONE is received (message includes the capacity):
  - store the current token ID (if contained in the message)
  - store the tokens (if contained in the message)
  - change the capacity
  - old assigned done = true
  - if reassignment has been started, call the reassign method

- When NEW_ DONE is received:
  - store the capacity
  - store the content
  - new assigned done = true
  - if reassignment has been started, call the reassign method

13. When OLD_CONTENT is received:
    14. old content done = true
    15. if reassignment has been started, call the reassign method

- Graph method for node notification of reassignment complete (already written)

    - decrement the counter
    - if the counter is zero, notify the PEP that reassignment is finished

- Variables for keeping track:
    - _state is an enumerated variable with the following values:
        - initial reassignment has not yet begun
        - awaitingOldAgents old assigned awaiting "done" messages from old agents
        - awaitingOldAssigned new assigned awaiting "done" message from old assigned
        - awaitingNewAssigned new agent awaiting "done" message from new assigned
        - awaitingOldContent old queue agent awaiting "done" message from old assigned
        - final all done with reassignment - report finished and reset variables for next reassignment
    - _numAgentsPending counter for the old assigned to keep track of incoming "done" messages
    - _oldAssignedDone boolean for new assigned to indicate receipt of "done" from old assigned
    - _newAssignedDone boolean for new agents to indicate receipt of "done" from new assigned

Modifications of existing code

For sending messages, only the code for the reassignment messages need be modified.

For receiving ordinary messages during reassignment:
- An old assigned that is not a new assigned receives a token message (TOKEN_MSG) from an old agent
    - forward the token message to the new assigned (OLD_TOKEN)
- An old assigned that is not a new assigned receives an agent capacity message (AGENT_CAPACITY) from an old agent
    - forward the capacity message to the new assigned
- An old agent that is not a new agent receives an ordinary content update message (ASSIGNED_CAPACITY)
    - ignore the message // the old agent may be new assigned or new idle
- Capacity changes // as is now the case, capacity change must be initiated via either an assigned place or an agent.
    - PEP calls capacity change for idle place (reassignment not in progress for the place) - do nothing
    - Idle place (reassignment not in progress for the place) receives capacity change message - do nothing
    - PEP calls capacity change for old assigned that is not new assigned - ignore it, because Joe says it will never happen
    - PEP calls capacity change for new assigned - notify new agents
    - PEP calls capacity change for old or new agent - send to respective old or new assigned
        - if the old assigned gets notification (by PEP call or by message from agent) before reassignment is started,
            - it will notify all old agents
            - when reassignment starts, it will send the modified capacity to the new assigned, which will notify the new agents

- if the old assigned gets notification after reassignment is started (and before it is done), it will forward to the new assigned, which will notify the new agents
- in any case, coherence of capacity between the new assigned and new agents is maintained.

Detailed algorithm for transition reassign method:

assign and activate the transition
report completion to the graph.

Detailed algorithm for place reassign method:

```
repeat forever {
// This will not really repeat forever, because on every pass the state changes, eventually returning in some state.
// If the return occurs in a pending state (waiting a "done" message that has not arrived), the function returns.
// When the "done" message arrives, the method is called again for the continuation of reassignment processing.
// When the "final" state is reached, the variables are reset for the next reassignment and completion is reported to the graph.
        switch on state:
        case initial
                remember old assignment and agents
                determine new assignment and agents

                if (was assigned ) {
                        if (will be assigned) { // was assigned, will be assigned
                                if (more than one upstream transition or old agent set != new agent set) {
                                        block
                                        if (a queue)
                                                send "content done" to old agents;
                                        state = awaiting "done" from old agents
                                }
                                else {
                                        state = final
                                }
                        }
                        else if (new agent) { // was assigned, will be agent
                                block
                                if (a queue)
```

```
                send "content done" to old agents;
            if (no old agents pending) {
                // collapse two messages into one without waiting for old agent "done" messages
                send "done" with token ID, capacity, and tokens to new assigned (OLD_ DONE)
                state = awaiting new assigned
            }
            else { //  some old agents pending
                send capacity and tokens to new assigned (OLD_TOKENS)
                state = awaiting old agents
            }
        }
        else { // was assigned, will be idle
            block
            if (a queue)
                send "content done" to old agents;
            if (no old agents pending) {
                // collapse two messages into one without waiting for old agent "done" messages
                send "done" with token ID, capacity, and tokens to new assigned (OLD_ DONE)
                state = final
            }
            else { //  some old agents pending
                send capacity and tokens to new assigned (OLD_TOKENS)
                state = awaiting old agents
            }
        }
}
else if (was agent) {
        if (will be assigned) { // was agent, will be assigned
            block
            send "done" to old assigned (AGENT_DONE)
            state = awaiting old assigned
        }
        else if (will be agent) { // was agent, will be agent
            if (more than one upstream transition OR old assigned != new assigned) {
                block
                send "done" to old assigned (AGENT_DONE)
```

```
                        state = awaiting new assigned
                    }
                    else { no more than one upstream transition AND old assigned = new assigned
                        state = final
                    }
                }
                else { // was agent, will be idle
                    block
                    send "done" to old assigned (AGENT_DONE)
                    if (a queue)
                        state = awaiting content done
                    else
                        state = final
                }
            }
            else (was idle) {
                if (will be assigned) {
                    state = awaiting old assigned
                }
                else if (will be agent) {
                    state = awaiting new assigned
                }
                else (will be idle) {
                    state = final
                }
            }

case awaiting old agents // old assigned waiting for done messages from old agents
            if (number of agents pending > 0) {
                return
            }
            else { // no pending agents - proceed
                if (will be assigned) { // was assigned, will be assigned
                    send "done" with C&C to new agents (NEW_DONE)
                    unblock
                    state = final
```

```
                }
                else if (will be agent) { // was assigned, will be agent
                        send "done" to new assigned (OLD_ALL_DONE)
                        state = awaiting new assigned
                else { // was assigned, will be idle
                        send "done" to new assigned (OLD_ALL_DONE)
                        state = final
                }
        }

case awaiting old assigned // was agent or idle, will be assigned
        if (not old assigned done) {
                return
        }
        else { old assigned done - proceed
                send "done" with C&C to new agents (NEW_DONE)
                unblock
                notify downstream transitions
                if (was agent AND a queue) {
                        state = awaiting old content
                }
                else
                        state = final
        }

case awaiting new assigned // new agent waiting for done message from new assigned
        if not new assigned done {
                return
        }
        else {
                unblock
                if ( was agent ) {
                        if (a queue)
                                state = awaiting old content
                }
                else {
```

```
                        state = final
                }
        }

    case awaiting old content
        if not content done { // old agent waiting for done message from old assigned
                              // Note:  this applies only for agents of queues
                              // content done is initialized off only for queue agents
            return
        }
        else {
            state = final
        }

    case final // final state for everybody - reset everything for next reassignment and notify complete
        reset the state variables
        reset the idle transition ports
        notify graph reassignment complete
        return

    case default
        error - throw an exception
}
```

Appendix D
Switch and Merge
Dick Stevens
January 20, 2001

Introduction

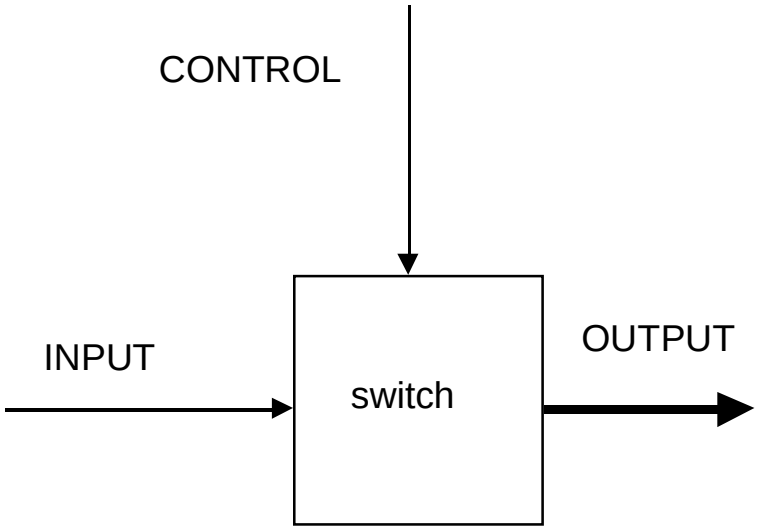
This document discusses included graphs for the switch and merge functions. The concepts of switch and merge have been described in the data-flow literature, and are recognized as being very powerful. At present I do not have a reference, however one place to look would be the doctoral thesis of Joe Buck, who was one of Edward Lee's graduate students at UC Berkeley.

For expediency and to demonstrate the versatility of PGM, we implement them as included graphs. For performance reasons, I recommend that each of these be implemented as a transition.

We begin by describing the functionality of switch and merge, followed by with an example of how they might be used. Finally, we give a more detailed description of an included graph for each.

Switch

We give a description of the switch in PGM terms. The following diagram shows the input ports and output ports of the switch:



There is a template parameter BASETYPE specifying the base type. There are also two GIPs, TOKENHEIGHT and OUTFAMSIZE.

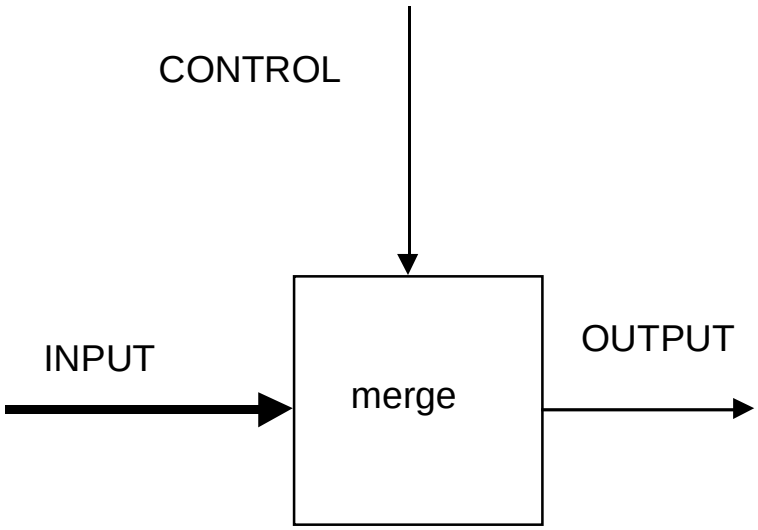
INPUT and CONTROL are single ports (i.e., height 0). OUTPUT is a height 1 family with size OUTFAMSIZE, indices ranging from 0 through OUTFAMSIZE – 1.

The INPUT and OUTPUT ports have the same token height and base type. We denote these by TOKENHEIGHT and BASETYPE, respectively. The token height and base type of the CONTROL port are 0 and int, respectively.

The token read at the CONTROL port is an integer that selects one of the OUTPUT ports, using the token value as an index. Specifically, if C is the token read from the CONTROL port, then OUTPUT[C] is the selected port. Thus, we would expect that 0 <= C < OUTFAMSIZE. If this condition is not met, then no OUTPUT port is selected. The token that is read at the INPUT port is then produced at the selected OUTPUT port. No tokens are produced at the non-selected OUTPUT ports. The respective tokens read at the CONTROL and INPUT ports are consumed.

Merge

We give a description of the merge in PGM terms. The following diagram shows the input ports and output ports of the merge:



There is a template parameter BASETYPE specifying the base type. There are also two GIPs, TOKENHEIGHT and INFAMSIZE.

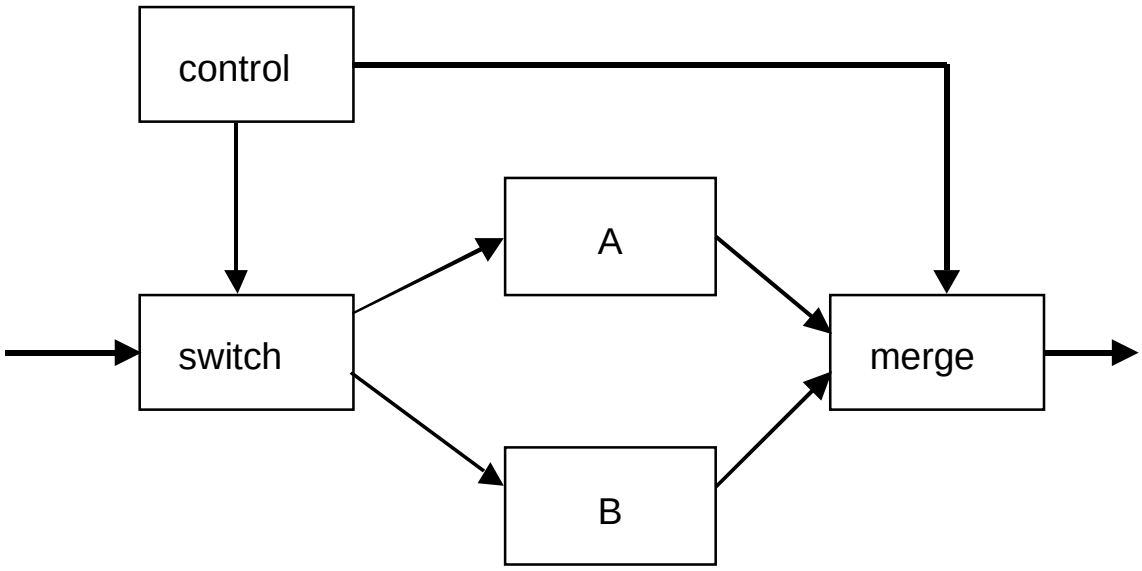
OUTPUT and CONTROL are single ports (i.e., height 0). INPUT is a height 1 family with size INFAMSIZE, indices ranging from 0 through INFAMSIZE – 1.

The INPUT and OUTPUT ports have the same token height and base type. We denote these by TOKENHEIGHT and BASETYPE, respectively. The token height and base type of the CONTROL port are 0 and int, respectively.

The token read at the CONTROL port is an integer that selects one of the INPUT ports, using the token value as an index. Specifically, if C is the token read from the CONTROL port, then INPUT[C] is the selected port. Thus, we would expect that 0 <= C < INFAMSIZE. If this condition is not met, then no INPUT port is selected. The token that is read at the selected INPUT port is then produced at the OUTPUT port. No tokens are read or consumed at the non-selected OUTPUT ports. The respective tokens read at the CONTROL and selected INPUT ports are consumed.

Example

This example illustrates a possible use for switch and merge:

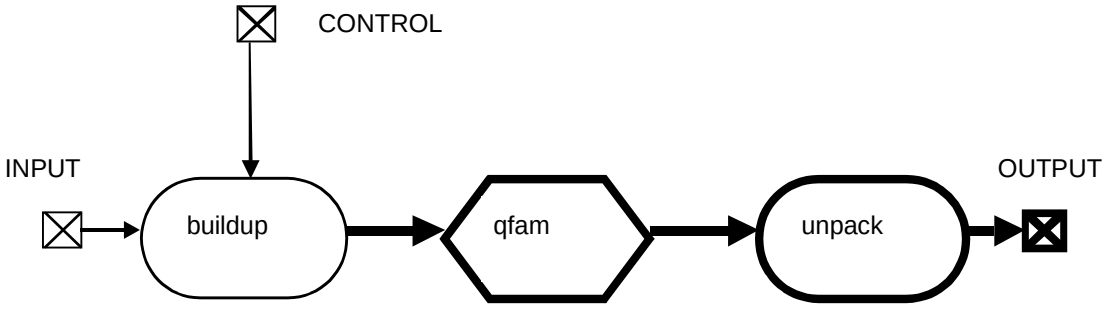


The control icon represents an included graph that produces integer tokens with the same value to each of the control ports of switch and merge. For the switch and merge icons the respective value for OUTFAMSIZE and INFAMSIZE is 2. Each of the icons A and B represents an included graph. If the control produces tokens with value 0 or 1, then the input put to the switch will be passed on to A or B, respectively, for processing. In either case, the result will be produced at the output of the merge.

Suppose that two control tokens 0 and 1 are produced successively by the control and that two tokens alpha and beta are read successively at the input of the switch. Then alpha will be processed by A, and beta will be processed by B with respective results A(alpha) and B(beta), possibly concurrently. Because of the control, A(alpha) will be produced before B(beta) even if B processes beta much faster than A process alpha.

Included Graph for switch

The following diagram shows the nodes for the included graph for switch:



The bold icons represent families. The unpack icon represents a height 1 family of OUTFAMSIZE unpack transitions. The qfam icon represents a height 1 family of OUTFAMSIZE queues, whose token heights and base types are TOKENHEIGHT and BASETYPE, respectively.

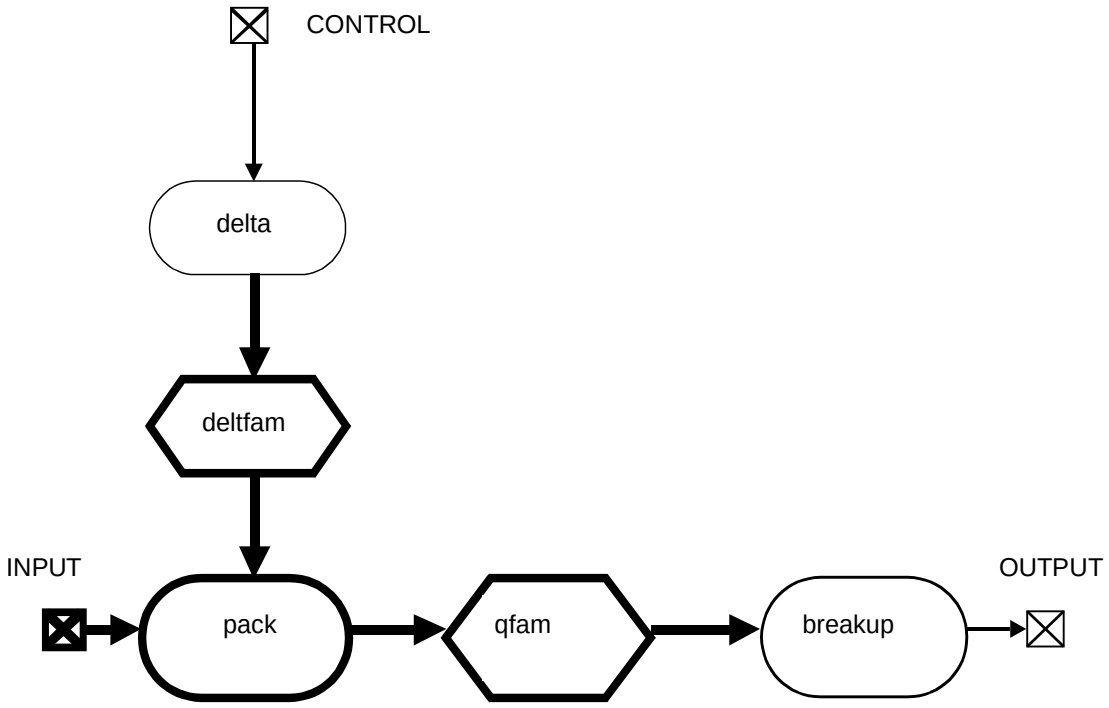
The buildup icon represents an ordinary transition. It has single ports INPUT and CONTROL, as specified above for the switch. It has a height 1 family of OUTFAMSIZE output ports, each with height TOKENHEIGHT+1 and base type BASETYPE. The transition statement reads the two input tokens (one from each of its input ports), and produces a token whose height is TOKENHEIGHT+2 as follows:

- First, it constructs a list of OUTFAMSIZE tokens with height TOKENHEIGHT+1. Let the tokens be denoted by T[i]; i = 0, 1, …, OUTFAMSIZE-1, and let C be the token read from its CONTROL port. Then each token is described as follows: If $i \neq C$, then T[i] is empty. T[C] has a single child equal to the value of the token read from the INPUT port.
- Then, the list of tokens T[i] are assembled to make up the desired output token.

Upon production, the family of OUTPUT ports of the buildup transition will automatically disassemble the token into respective tokens T[i] with height TOKENHEIGHT+1, each to be stored in the respective qfam queue. When the unpack transitions execute, if $i \neq C$, nothing will be produced, because T[i] is empty. If i = C, then the token T[C], having a single child with the desired value, will produce a single token, as specified for the switch.

Included Graph for merge

The following diagram shows the nodes for the included graph for merge:



The bold icons represent families. The pack icon represents a height 1 family of INFAMSIZE unpack transitions. The qfam icon represents a height 1 family of INFAMSIZE queues, whose token heights and base types are TOKENHEIGHT and BASETYPE, respectively.

The delta icon represents an ordinary transition. It has a single input port IN and a height 1 family OUT of INFAMSIZE output ports. Both IN and OUT have base type int. The transition statement produces a height 1 token T of integers with values T[C] = 1, and T[i] = 0 for $i \neq C$. This is the well-known Kronecker Delta function. Upon production from the delta transition, the output port family will disassemble T, so that each deltfam[i] (token height = 0, base type is int) gets a 0 if $i \neq C$ or a 1 if i = C.

The pack icon represents a family of INFAMSIZE pack transitions. Each deltfam queue is connected to the READAMOUNT port of the respective pack. Thus, the selected pack transition will have a read amount of 1, and all the others will have a read amount of 0. Every pack transition will produce a token with height 1. The selected pack's output token will have a single child equal to the token read from its input port. All other pack's output tokens will be empty.

The breakup icon represents an ordinary transition. Its input port is a height 1 family with INFAMSIZE ports, each with token height TOKENHEIGHT+1 and base type BASETYPE. Its output port is single port with token height TOKENHEIGHT and base type BASETYPE. When the breakup transition executes, its family inport assembles the tokens produced by all the pack transitions into a single token with height TOKENHEIGHT+2, which the transition statement processes. The transition statement first disassembles this token into a list of tokens T[i], each with height TOKENHEIGHT+1. Note that T[C] has a single child whose value is the desired output. All other T[i]; $i \neq C$, will be empty. Thus, the transition statement finds the T[i] that is not empty and produces its child to the output.